AJ

Bell Laboratories Cover Sheet for Technical Memorandum

The information contained herein is for the use of employees of Bell Laboratories and is not for publication. (See GEI 13.9-3)

Title- **LIL Reference Manual**

Date- **June 19, 1974**

TM- **74-1352-8**

Other Keywords- **PDP-11**
Implementation Languages

Author	Location	Extension	Charging Case- **39394**
P.J. Plauger	**MH 7C-205**	**X3644**	Filing Case- **39394**

ABSTRACT

LIL is a Little Implementation Language for PDP-11 computers, suitable for writing system level code or in any situation where assembly-language coding is traditionally called for. A LIL compiler is available for use under the UNIX operating system. The object code produced is compatible with, and may be freely intermixed with, that produced by the UNIX assembler, Fortran, or C compiler.

This document is a reference manual for the LIL language. A knowledge of machine level coding on the PDP-11 is assumed, and some knowledge of UNIX operating procedures is required to use the compiler. A tutorial introduction to programming in LIL is provided in TM 74-1352-6.

Pages Text 11	Other 0	Total 11	M. J. Melchner
No. Figures 0	No. Tables 0	No. Refs. 0	

COMPLETE MEMORANDUM TO

CORRESPONDENCE FILES

OFFICIAL FILE COPY
PLUS ONE COPY FOR
EACH ADDITIONAL FILING
CASE REFERENCED

DATE FILE COPY
(FORM E-1328)

10 REFERENCE COPIES

SOYD,GARY D
+BUCHSSAUM,S J
CHRISTENSEN,C
+CLOGSTON,A M
+CONDON,J H
CUTLER,C CHAPIN
FREENY,S L
GILLETTE,DEAN
GIORDANO,PHILIP P
+HANNAY,N B
+JENSEN,PAUL D
KEEFAUVER,W L
+KLEIN,MISS R L
+KOLETTIS,N J
LIMB,J O
LUCKLAWA,HEINZ
MCDONALD,R S
+MILLER,S E
NINFE,WILLIAM H
NOOLANDI,J
+OMOHUNDRO,WAYNE E
PATEL,C K N
PRIM,ROBERT C
+REID,ROBERT E JR
+ROBERTS,CHARLES S
SLICHTER,W P
TEWKSBURY,S K
THOMPSON,JOHN S
TILLERSON,L C
WOLONTIS,V MICHAEL
YAMIN,MRS E E
+YOUNG,JAMES A
32 NAMES

COVER SHEET ONLY TO

CORRESPONDENCE FILES

4 COPIES PLUS ONE
COPY FOR EACH FILING
CASE

ABRAHAM,STUART A
ACKERMAN,A F
AHO,A V
AHRENS,RAINER B
ALCALAY,DAVID
ALLES,HAROLD G
AMRON,I

ANDERSON,M M
ARMBRUSTER,MISS M E
ARNOLD,GEORGE W
ARNOLD,S L
ATAL,S S
BADURA,DENNIS C
BALDWIN,GARY L
BARTLETT,MARY S
BASELI,RICHARD J
BAUER,MISS H A
BAUGH,C R
BENGLANG,G DAVID
BERING,D E
BERNSTEIN,LAWRENCE
BEYER,JEAN-DAVID
BILOWOS,RICHARD M
BIREN,MRS IRMA B
BISHOP,MISS V L
BLINN,JAMES C
BLUE,J L
BLY,JOSEPH A
BODEN,F J
BOHACREVSKY,I O
BOWYER,L RAY
BOYCE,W M
BRANDT,RICHARD B
BREWER,F A
BROWN,J STANLEY
BROWN,WILLIAM R
BULLEY,RAYMOND M
BURR,STEFAN A
BUTSIEN,PAUL E
CAMPBELL,STEPHEN T
CARAWAY,R E
CASPERS,MRS BARBARA E
CAVINESS,JOHN D
CEMASHKO,FRED
CHAMBERS,J W
CHAMBERS,MRS B C
CREN,EDWARD
CHEN,STEPHEN
CHERRY,MS L L
CHODROW,MARK M
CLAYTON,DANIEL F
CLIFFORD,ROBERT M
COBEN,ROBERT M
COHEN,HARVEY
COLE,LOUIS M
COLLIER,ROBERT J
COOPER,A E
CORASICK,MISS M J
COULTER,J REGINALD
COURTNEY PRATT,J S
CRUME,LARRY L
D ANDREA,MRS LOUISE A
DAVIS,R L JR
DEUTSCH,DAVID N
DICKMAN,B N
DIMINO,L A
DINNICK,JAMES O
DOLOTTA,T A
DRAKE,MRS L
DRYDEN,JOHN J

EDWUNDS,T W
EIGEN,D
ELLIOTT,R J
ELY,T C
ENGEL,F W
ESSERMAN,ALAN R
FABISCH,MICHAEL P
FANOO,GEORGE A
FELDMAN,STUART I
FELS,ALLEN M
FIGLIUZZI,MISS M E
FISCHER,H L
FLANAGAN,J L
FLEISCHER,HERBERT I
FONG,KENNETH T
FORT,JAMES W
FOX,PHYLLIS
FOX,R T
FOY,J C
FRANK,MISS A J
FRANK,RUDOLPH J
FRASER,A B
FREEMAN,K GLENN
FRETWELL,LYMAN J
FROST,H BONNELL
FULTON,ALAN W
GARCIA,R F
GATES,G W
GAY,FRANCIS A
GEARY,M J
GELBER,MISS CHERON L
GEPNER,JAMES R
GEYLING,F T
GIBB,KENNETH R
GILBERT,MRS HINDA S
GIMPEL,JAMES F
GITMENS,JOHN A
GLASSER,A
GLUCK,F
GOGUEN,MS NANCY
GOLAGEK,MISS R
GOLDSTEIN,A JAY
GORMAN,JAMES E
GRAHAM,R L
GREENSPAN,S J
GROSS,ARTHUR G
GUERRIERO,JOSEPH R
HAFER,E H
HALL,ANDREW D JR
HALL,MILTON S JR
HALL,W G
HAMMING,R W
HANSEN,P J
HARASYMIW,J
HARKNESS,MRS CAROL J
HARRINGTON,T DORSEY
HARRISON,NEAL T
HARUTA,K
HAUSE,A D
HEATH,SIDNEY F III
HEMMETER,RICHARD W
HEMMING,C JR
HERGENHAN,C S

HEROLD,JOHN W
HESS,MILTON S
HINES,MISS P E
HOLTMAN,JAMES P
HONIG,W L
HOTT,WILLIAM F
HO,J C
HUNNICUTT,CHARLES F
IFFLAND,FREDERICK C
IPPOLITI,O D
IRVINE,M M
IVIE,EVAN L
JACKOWSKI,D J
JACOBS,H S
JAKIELSKI,C E
JAMES,DENNIS B
JESSOP,WARREN B
JOHNSON,STEPHEN C
JOHNSTON,W DEXTER JR
JOYNT,L
JUDICE,CHARLES N
KAPLAN,A M
KAPLAN,ROBERT
KAYEL,R G
KEARNEY,ROBERT
KELLY,L J
KENNEDY,ROBERT A
KERRIGHAN,BRIAN W
KERTZ,DENIS R
KILLMER,JOHN C JR
KNOWLTON,KENNETH
KNUDSEN,DONALD B
KORNEGAY,R L
KREIDER,DANIEL M
KRUSKAL,JOSEPH B
LEE,WILLIAM C Y
LEGENHAUSEN,S
LESK,MICHAEL E
LESK,MS ANN B
LIEBERT,THOMAS A
LINDERMAN,J
LINNEWEB,LOUIS H
LIN,SREN
LOEGREN,M M
LUMMIS,ROBERT C
LUTZ,KENNETH J
MADDEN,MRS D M
MAHLER,G R
MALCOLM,J A
MALLOWS,COLIN L
MARSH,MISS M
MAUNSELL,HENRY I G
MC CABE,PETER S
MC CRACKEN,MARYLOU
MC CUNE,B F
MC EOWEN,JAMES R
MC GILL,ROBERT
MC GONEGAL,MISS C A
MC ILROY,M DOUGLAS
MC MULLEN,E C
MC RAE,JEAN E
MC TIGUE,G E
MENIST,DAVID S

MENON,P R
METZ,ROBERT F
MILLER,ALAN R
MILLS,MISS ARLINE D
MOLINELLI,JOHN J
MOLTA,J W
MORGAN,DENNIS J
MORGAN,S P
MORRIS,ROBERT
MORR,PHILLIP L
MURATORI,RICHARD D
NERRLICH,W R
NELSON,N-P
NORTON,ROBERT D
NOWITZ,D A
O CONNELL,T F
O SREA,W T
O SULLIVAN,JOHN A
OLSEN,RONALD G
OPPERMAN,D C
ORSANNA,J F JR
OTTO,JOHN R
OWENS,MRS G L
PALM,RICHARD C JR
PEARLMAN,R
PFNNING,T
PFRITSKY,MARTIN M
PETERSON,RALPH W
PETERSON,T G
PFISTER,ROBERT G
PILLA,MICHAEL A
PINSON,ELLIOT M
PIRZ,FRANK
PITTS,CHARLES J
POLLAK,HENRY O
POPPER,C
RAACK,GERALD A
REED,R C
REHERT,ALLEN F
RIDDLEBERGER,C O
RIDDLE,J G
RITACCO,J F
RITCHIE,DENNIS M
ROCHFORD,M J
RODIRAN,MRS PATRICIA A
RODRIGUEZ,ERNESTO J
ROSENTHAL,CHARLES W
ROSLER,LAWRENCE
ROVEGNO,MRS HELEN D
RYDER,J
SATZ,L R
SCHRYER,N L
SCHURTER,W R
SCHWENKER,JOHN F
SFARS,EDWARD R
SEKINO,W F
SEMMELMAN,C L
SHANK,J C
SHANK,R A
SHIPLEY,EDWARD N
SHORTER,J W
SIDOR,DAVID J
SINOWITZ,NORMAN R

+ NAMED BY AUTHOR

> CITED AS REFERENCE SOURCE

...
317 TOTAL

MERCURY DISTRIBUTION.....

COMPLETE MEMO TO:
10-EXD 13-DIR 135-DPH

COVER SHEET TO:
127
COPLAS = COMPUTING/PROGRAMMING LANGUAGES/ASSEMBLY

RADY,J E; MH 7B201;

TM-74-1352-8 TOTAL PAGES 11

TO GET A COMPLETE COPY:

1. BE SURE YOUR CORRECT ADDRESS IS GIVEN ON THE OTHER SIDE.
2. FOLD THIS SHEET IN HALF WITH THIS SIDE OUT AND STAPLE.
3. CIRCLE THE ADDRESS AT RIGHT. USE NO ENVELOPE.

PLEASE SEND A COMPLETE COPY TO THE ADDRESS SHOWN ON THE OTHER SIDE
NO ENVELOPE WILL BE NEEDED IF YOU SIMPLY STAPLE THIS COVER SHEET TO THE COMPLETE COPY.
IF COPIES ARE NO LONGER AVAILABLE PLEASE FORWARD THIS REQUEST TO THE CORRESPONDENCE FILES.

Bell Laboratories

Subject- **LIL Reference Manual**

Date- **June 19, 1974**

From- **P. J. Plauger**

TM- **74-1352-8**

MEMORANDUM FOR FILE

LIL is a Little Implementation Language for the PDP-11 family of computers. It is *little* in the important sense that there are very few constructs and combining rules, and fewer exceptions. It is an *implementation language,* which means that it is possible to express any of the code needed to implement an operating system, including I/O drivers, interrupt routines, and transfer tables. Nevertheless it remains a moderately high-level language, with the improved readability and reliability that one expects from avoiding assembly-language level coding.

This document is a reference manual for LIL. It provides a comprehensive, and reasonably precise, description of the language, but makes no attempt to introduce concepts in a tutorial order.

1. Invocation

The LIL compiler is invoked by

lc args

where **args** are source files if they end in **.l**, flags if they begin with **−**, and object files otherwise. If more than one source file is specified, or if the **−c** flag is present, object code for each file whose name ends in **.l** is written on a file with a similar name, only ending in **.o**.

Unless **−c** (compile only) or **−P** (preprocess only) is present, all object files are presented to the loader for binding, along with the standard C libraries. Profiling may be invoked with **−p** (see cc (I), prof (I) ld (I), monitor (III) in the UNIX manual).

2. Preprocessor

If the first character of a LIL source file is a **#**, a string-oriented preprocessor is invoked. This scans the source for lines beginning with a **#** and recognizes the following non-empty command lines:

 # include 'pathname' — The file specified by **pathname** is included in the source in place of the command. Line numbering of the remainder of the source file is unaffected. For the sake of diagnostics, all lines in the included file have the same line number.

 # define SYMBOL string — The **string** of characters on the remainder of the line is remembered, along with the **SYMBOL** name, and the defining line is blanked out. All subsequent occurrences of **SYMBOL** are replaced by **string**, surrounded by blanks. **SYMBOL** is a maximal string of letters, digits, dot '.', and underline '_', beginning with a non-digit.

The **-P** option on the invocation line may be used to cause preprocessor output for each file whose name ends in .l to be written on a file with a similar name, only ending in .i. PL/I style comments are permitted on preprocessor command lines.

3. Lexical analysis

The source file is treated as a series of *tokens*, possibly separated by *white space*, which consists of one or more blanks, tabs, newlines, and/or comments. Comments are arbitrary strings beginning with a % and ending with the next newline. Except for its important function of separating tokens, white space is ignored, and so should be used freely to emphasize the logical structure of the code.

Tokens are one of the following strings of characters:

identifier — one or more letters, digits, '.', and/or '_', beginning with a non-digit. Only the first eight characters are used to distinguish identifiers.

octal number — a 0, followed by any number of digits. Only the low order sixteen bits of the value represented by the string are retained.

decimal number — a non-zero digit, followed by any number of digits. If the value represented by the string is greater than 32767, its value as a token is uncertain.

character string — a pair of single quotes enclosing any number of characters. Each character may be any valid ascii code except newline or single quote. A \ is combined with the character following to form a single character; the following mapping occurs:

\0 becomes	000	NUL
\a	006	ACK
\e	004	EOT
\n	012	NL
\p	033	PRE (ESC)
\r	015	CR
\t	011	TAB

All other characters following \ are mapped into themselves (including \, newline and single quote).

operator — one or more of the characters from the set '=+-<>?*/&|!~^'. The pairs |^ and ~! are each interchangeable.

punctuation — A character from the set ';,{}[]()\'. The pairs {(and)} are each interchangeable, but it is recommended that the stylistic usage followed in this description be retained wherever possible. The equivalences are provided to ease typing readable LIL programs on a 64-character terminal.

Other ascii characters not shown here will produce a diagnostic message and be skipped.

4. Keywords

Certain identifiers are predefined as *keywords*, i.e. tokens with special syntactic meaning. These identifiers are:

break	**goto**	**rts**
byte	**if**	**sizeof**
continue	**jsr**	**sys**
do	**local**	**while**
else	**mem**	**word**
extern	**reg**	

Keywords may not be redefined in local regions, unlike other identifiers.

It is possible, however, to remove an identifier from the keyword class, once and for all, by means of the escape \. Any occurrence of a keyword immediately preceded by a \ is treated as a reference to a hitherto undefined identifier of that name. This feature should only be used when absolutely necessary, as when interfacing to a foreign language routine, and then only near the end of a LIL source file.

With this understanding, the term *identifier* will be used hereafter to mean any non-keyword or a keyword that has been properly escaped.

5. Locatives

Identifiers, octal numbers, decimal numbers, and character strings no longer than two characters are the simplest forms of *locatives,* which are used to specify the location of operands for generating machine instructions, or which are the actual operands of compile-time operations. A locative has a *value,* which may be defined or undefined, absolute or relocatable with respect to some *bias.* Each locative also has a *type,* which corresponds closely to a valid PDP-11 addressing mode or specifies a *condition,* a *size* in bytes, and the optional attributes *external* and *byte.* Octal numbers, decimal numbers, and short character strings, for instance, are by default type *immediate,* and have values which are defined and absolute.

Unary operators are provided to alter type and other attributes. In addition, a number of predefined identifiers are provided from which type can be inherited. The basic types are:

register — The value of the locative must be defined absolute between 0 and 7, inclusive. The operand is the contents of the general register specified by the value.

memory — The value of the locative is taken as an address in memory at which the operand is to be found. The value may be absolute, to refer to special addresses used by the hardware, or relocatable, to refer to addresses within the user program. If the value is undefined, it is assumed that a definition will be provided later in the program text (forward reference) or at load time from another object module (undefined external).

immediate — The value of the locative itself is to be the operand. Immediates may also be undefined or relocatable.

More elaborate types are built from these primitives, or from expressions whose resultant type is register, memory, or immediate. Square brackets are used to indicate indexing or indirection; in conjunction with additional unary operators the remaining types are specified:

autodecrement — The notation **--[rexp]**, where **rexp** indicates a register locative or an expression whose resultant type is register, specifies autodecrement addressing using the register determined by the value of the locative or expression.

autoincrement — The notation **[rexp]++** specifies autoincrement addressing in an analogous fashion. **++** is the only suffix operator defined in LIL.

indexing — The notation **x[rexp]**, where **x** is a locative of type memory or immediate, specifies indexing on the register determined by **rexp**. The value of the locative is the same as **x**, while the index register used becomes part of the type information.

indirection — The notation **[loc]**, where **loc** is any locative that does not itself involve indirect addressing, specifies indirect addressing. By special dispensation, however, **[[rexp]]** is permitted; it is translated into **[0[rexp]]**. This notation results in a family of types: indirect on register, indirect on memory, indirect on autodecrement, etc.

The final type which locatives may assume is *condition,* which specifies what state of the condition code must obtain for a given test to succeed. Such locatives are always defined absolute, and have mystical values which closely resemble the appropriate op codes for conditional branches. Conditions usually arise from relational expressions or tests and seldom need be dealt with explicitly.

6. Predefined identifiers

LIL begins each compilation with a number of predefined identifiers which are often of use in writing code.

All eight registers may be referenced by their conventional names:

```
r0      r4
r1      r5
r2      sp
r3      pc
```

The symbol dot '.' is updated at the end of each statement to point to the next location at which code is to be generated. Similarly, setting dot to a new value causes loading to be diverted to the place specified. The location counters for the *bss*, *text*, and *data* sections (see ld(I) in the UNIX manual) are named **.bss**, **.text**, and **.data**, respectively. In addition, the absolute location counter **.abs** is provided for convenience, to remember the latest absolute value dot may have assumed. All of these locatives are of type memory.

Conditions corresponding to testing just one of the condition code bits may be specified by using the predefined:

```
carry
minus
oflow
zero
```

These correspond to the C, N, V, and Z bits, respectively. Unconditional tests may be written with

```
false
true
```

in an obvious fashion.

All of these predefined identifiers may be superseded by local definitions, making the original versions unreachable but still not interfering with their maintenance by the compiler. There is one additional predefined identifier, however, which serves as a flag; its latest edition is always consulted by LIL. This is **.temp**, which is used to specify whether or not the top element of the stack **[sp]** is to be used to hold an argument on function calls. If the standard function call notation is used, therefore, this must be considered a reserved identifier at all times.

7. Expressions

Data manipulation instructions are generated by writing *expressions* involving locatives and binary infix operators:

```
r0 = x;          % causes r0 to be loaded from x
r0 + 1;          % causes r0 to be incremented
r0 = x + 1;      % load then increment r0
```

The interpretation of the last statement differs markedly from that used in most languages. All infix operators are of equal precedence in LIL, evaluation proceeds strictly left to right. The left operand is retained for use with each subsequent operator, permitting the shorthand shown in the example. (The third line is equivalent to the first two.)

Parentheses may be used to modify this strictly left to right order of evaluation:

```
r0 = (x + 1);   % increment x, then move x into r0
```

The rule is: evaluate the left operand, performing any calculations inside parentheses or brackets, then do the same for the right operand, then perform the operation specified between the two

operands. The leftmost locative inside parentheses becomes the locative to use as operand.

All unary operators bind tighter than binary operators, as does the specification of indexing or indirect addressing with square brackets. Individual unary and binary operators will be described in a later section.

8. Compile time

To perform arithmetic with symbols, without generating code, and to define identifiers, one writes what look like normal expressions, surrounded by double quotes. Thus

```
"x = r0";          % defines x
"r0 + 1" = x;      % loads x into r1
```

Compile-time expressions are actually evaluated in a completely different context than run-time expressions, using the values of the locatives themselves as operands. Most operators have compile-time definitions analogous to their meanings at run time. Unless it contains an explicit assignment operator (= or ->), a compile-time expression is computed without changing the value of any identifiers.

Compile-time binary operators will also be tabluated later on.

9. Statements

An expression, terminated by a semicolon, is one of the simplest statements permitted in LIL. An immediate or memory locative standing alone, followed by a semicolon, is treated as a request to generate one word of code whose value is the value of the locative. Character strings of any length may be used in this context to generate a series of code words; the last is padded with a null byte if the number of characters is odd. It is also permissible to write just a semicolon to specify a null statement.

The simplest conditional statement is:

if (test) statement;

test may be any locative, or a conditional expression containing parentheses, the prefix *not* operator ~, the infix *and* operator &&, and/or the infix *or* operator ||. Conditional expressions are evaluated left to right, following the same rules as for normal expressions, except that the right operand of && will not be evaluated when the left operand is false, and the right operand of || will be likewise skipped when the left operand is true. Moreover, && binds tighter than || for the sake of determining the truth value of the entire test. If a locative is not a condition, it is replaced by that condition obtained by testing if the locative is nonzero.

The not operator, of course, reverses the sense of any condition. Care must be taken, however, to ensure that its operand is unmistakably a condition, for ~ applied to an immediate merely ones-complements its value.

The **if** causes the controlled **statement** to be skipped if the test is not met, otherwise the statement is obeyed. To specify an alternate action when the test fails, write

if (test) statement; else statement;

The **else** part is skipped over if the first statement is executed. The first statement may be another **if**, in which case there is a possibile ambiguity in pairing subsequent **else** clauses. This ambiguity is arbitrarily removed by binding each **else** clause to the innermost 'unelsed' **if**.

Loops are specified by

while (test) statement;

if the test should occur at the top, or by

do statement; while (test) statement;

if the test should occur at the end"if the test should occur somewhere in the middle of the loop or at the end (second statement is null). Both forms exit when the test fails and contain an implicit branch back to the beginning from the end of the form.

10. Groups

Conditional and loop statements can be made much more powerful by having them control *groups*. A group is one or more statements in succession surrounded by braces:

```
if (r0 < 0)                 % if r0 is negative
    {r0 =- r0;              % negate r0
    r1 + 1; }               % and count
```

No semicolon is used after a group, nor is the semicolon left off the last statement in the group. Now the **if** will skip two statements if the test fails. Groups may, of course, contain other groups to any depth of nesting.

A second form is the *labelled group:*

```
label{ statement;
       statement;
       ...; }
```

where **label** is an identifier. In addition to grouping statements for control purposes, this form also delimits a *local region*, in which identifiers may be redefined without clashing with other usage, and causes the label to be defined as type memory with the value of the location counter at the start of the group (although the definition actually occurs at the close of the group).

Two control statements work in conjunction with loops or labeled groups. The forms

```
break;
continue;
```

cause control to be transferred out of the innermost containing loop or labeled group, or back to its top, respectively. And the forms

```
break label;
continue label;
```

perform similarly, except that all containing loops or groups whose labels do not match the label in the statement are ignored. Instead, control is transferred out of, or to the top of, the innermost containing labeled group whose label matches.

11. Function call and goto

A C compatible function call is provided:

```
fun();                      % no arguments
fun(arg, arg, arg, ...);
```

where **arg** is any locative or expression except a condition. Any argument expressions are evaluated left to right, then the arguments are moved onto the stack *right to left*. The function is called with a

```
jsr pc,*$fun
```

instruction and the arguments are popped off the stack.

The latest edition of the flag variable **.temp** is consulted on each call with arguments to determine whether to use the top element of the stack **[sp]** to hold the rightmost argument. The flag is initially zero, indicating that **[sp]** is not to be used.

Each function call becomes a locative of type register and value zero.

Control can be transferred to an arbitrary destination by

```
goto loc;
```

where **loc** is any locative that may be used with a **jmp** instruction (a **br** will be generated instead, wherever possible). It should not be necessary to use **goto** except in unusual circumstances.

12. Declarations

Identifiers are made implicitly local to the first region in which they are referenced. If, by the close of that region, no definition is provided, the identifier is promoted to the next containing region. Identifiers undefined at the close of the source file are published as *undefined externals.* To explicitly create local identifiers, use

```
"local ident, ident, ident, ...";
```

where **ident** is an identifier, whose size may optionally be specified by writing

```
ident sizeof loc
```

for **loc** defined absolute immediate. Default size is two bytes. If these **idents** are not defined by the close of the current local region, a diagnostic is produced.

Similarly, one can declare that certain identifiers are to be made known to other compilations, if defined, or are to be obtained from other object modules, if undefined, by writing

```
"extern ident, ident, ident, ...";
```

An identifier explicitly declared local and not defined must also appear in an external declaration.

A useful variant of the **local** statement is

```
"loc(ident, ident, ident, ...)";
```

For each **ident**, left to right, this

1) declares **ident** to be local and defines it to have all the same attributes, except size, as **loc**.
2) increments the value of **loc** by the size of **ident** (if **loc** is an identifier this constitutes a redefinition).

The final size of **loc** is the sum of the sizes of the **idents**.

13. Load Control

LIL begins loading in the *text* section, but permits code to be generated also in the *data* section, by redefining dot, provided no attempt is made to back up over generated code.

```
". = .data";            % switch to data area
x{-1; }
y{0; 0; 0; }
". = .text";            % switch back
```

By the rules for labeled groups, however, such a temporary diversion may also be written as

```
". = .data"{
x{-1; }
y{0; 0; 0; }
        }
```

provided x and y have been previously referenced or declared.

Dot can be set to any absolute value, or to point anywhere in the *bss* section, so long as no attempt is made to generate code there.

UNIX treats all undefined external references with nonzero size as labeled common blocks, and refuses to satisfy such references with *text* symbols. LIL will set the size attribute to zero on any identifier used in a function call. Nonstandard entries which are not defined in the current file, however, should be declared by

```
"extern entry()";
```

to set the size attribute of **entry** to zero and declare it external.

14. Unary operators

Unary operators have the same meaning at compile-time as at run-time, for none of them generate code (except **=**). Instead, they modify *references* to symbols or expressions, by changing their type, value, or other attributes. For convenience the following symbols are defined:

! — non-byte memory reference.

m, n — absolute (i.e. unrelocatable) immediate.

r — register.

v, w — any non-byte address type.

x, y — any PDP-11 address type, including byte.

The unary operators are:

x++ — converts type register indirect to type autoincrement.

--x — converts type register indirect to type autodecrement.

=n — compiles the absolute immediate or string **n** into the *data* section, appending a zero word if the last byte is not null. Result is type immediate, value is address of start of the allocated string.

&x — converts **x** to type immediate, removes byte attribute.

mem x — converts **x** to type memory, removes byte attribute.

reg x — converts **x** to type register, removes byte attribute. **x** must be defined absolute with 0 <= value <= 7.

word x — removes byte attribute.

byte x — attaches byte attribute.

-n — negates absolute immediate **n**.

~n — complements absolute immediate **n**.

rts r — converts type register **r** to an immediate with value of **rts r** instruction.

jsr r — converts register **r** to an immediate with value of **jsr r,*$** instruction. (Should be followed by address of routine.)

sys n — converts immediate **n** to an immediate with value of **sys n** instruction.

sizeof x — converts **x** reference to an absolute immediate with value equal to size attribute of **x** reference.

lll n — shifts absolute immediate **n** left 8.

15. Compile-time binary operators

x>=y, x>y, x==y, x<y, x<=y, x~=y — result is a condition with value **true** if the relation obtains, otherwise **false**. Both x and y must have the same relocation bias. Comparisons are all on signed quantities.

x=y, y->x — x is defined to have the same type, value, and other attributes as the y reference. x must be previously undefined (unless it is dot or one of the four location counters).

x=-n — x is defined as the negative of the absolute symbol n.

x=~n — x is defined as the complement of the absolute symbol n.

The type of x is left unchanged in the following:

x+y — result is the sum of x and y. At most one of the two may be relocatable or undefined.

x-y — result is the difference of x and y. If y is relocatable or undefined, it must have the same bias as x.

n~m — result is n equivalence m (not exclusive or).

n~~m — result is n exclusive or m (not equivalence).

n|m — inclusive or.

n&m — and.

n&~m — same as n&(~m).

nm** — n is shifted left (if m>0) or right (if m<0) by |m|.

n*m — multiplication.

n/m — division.

n|||m — same as (n&0377) | (m**8) [packs bytes].

x sizeof n — result is x with size n.

16. Run-time binary operators

If x and y are not both byte or both word, then one of them must be a non-byte register or immediate. The instruction generated will then be byte mode if either operand is byte. The following is a metalinguistic description of the decisions made by LIL in producing code for each operator.

```
x>=y, x>y, x==y, x<y, x<=y, x~=y        [compare signed]
x>>=y, x>>y, x<<y, x<<=y                [compare unsigned]
        if (y==0 && cc set on x)        do nothing
        else if (y==0)                  tst(b) x
        else if (x==0)                  tst(b) y
        else                            cmp(b) x,y

x=y,
y->x    if (y==0)                       clr(b) x
        else if (x=={carry oflow zero minus} && y=={true false})
                                        setx or secx
        else if (y==minus)              sxt x
        else                            mov(b) y,x

x=-y    if (y==0)                       clr(b) x
        else if (x==y)                  neg(b) x

x=~y    if (x=={carry oflow zero minus} && y=={true false})
                                        secx or setx
```

```
          else if (x==y)          com(b) x

x+y       if (y==1)               inc(b) x
          else if (y==carry)      adc(b) x
          else                    add y,x

x-y       if (y==1)               dec(b) x
          else if (y==carry)      sbc(b) x
          else                    sub y,x

x|y                               bis(b) y,x

x&n                               bic(b) $!n,x

x&~y                              bic(b) y,x

x^^r                              xor r,x

r*y                               mul y,r

r/y                               div y,r

x?y       if (y==0)               tst(b) x
          else                    cmp(b) x,y

x?&y                              bit(b) x,y

x<>n      if (n==1)               rol(b) x
          else if (n== -1)        ror(b) x

x<*>n     if (n==8)               swab x
          else if (x odd reg)     ashc n,x

x**n      if (n==1)               asl(b) x
          else if (n== -1)        asr(b) x
          else if (x a reglster)  ash n,x

r***n     ashc n,r
```

Both the **sizeof** and **|||** operators are also defined at run-time and have the same effect as at compile-time. They do not directly cause code to be generated.

The compiler is aware of what is happening to the condition code most of the time. It knows, for instance, that **swab** and function calls do not leave the code in a state implied by the language, and so will generate a **tst** if the result of either is to be used in a test. The PDP-11 is somewhat whimsical about the setting of the carry bit, however, and LIL makes no real attempt to second guess the machine. (It makes a difference whether you add 1 or 2 to something, for instance.) It is always a good idea to be very careful when testing for special conditions.

17. Conditional Compilation

If the **test** in any conditional or loop statement is unconditionally true or false, no code is compiled for the test. If the test is false, in fact, no code is compiled for the statement controlled by the test (the branch back to the top of a loop is also omitted). In the **if-else** form, one and only one of the two statements is compiled. Compile-time parameters, in conjunction with compile-time relational expressions, can thus be used to cause selective generation of code.

18. Pedigree

LIL descends from an implementation language for the GTE TEMPO-I, written by P.D. Jensen and A.G. Fraser. Compile-time notation and the rigorous left-to-right expression evaluation are carryovers. The current syntax is heavily influenced by that of the language C, designed by D.M. Ritchie; the preprocessor and invocation control are a direct steal.

All compilation decisions are made in one pass of a simple syntax-directed translator, written by the author. Control tables for the translator were produced by Steve Johnson's compiler-compiler YACC, working from a 50-production grammar. Doug Bayer wrote the last pass, which assembles code and tables into standard UNIX object format.

P J Plauger

MH-1352-PJP P.J. PLAUGER